Programming Sensor Networks Using REMORA Component Model

Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, Frank Eliassen

Programming Sensor Networks Using REMORA Component Model

Amirhosein Taherkordi[1], Frédéric Loiret[2], Azadeh Abdolrazaghi[1], Romain Rouvoy[1,2], Quan Le-Trung[1], and Frank Eliassen[1]

[1] University of Oslo, Department of Informatics
P.O. Box 1080 Blindern, N-0314 Oslo
{amirhost,azadeha,rouvoy,quanle,frank}@ifi.uio.no
[2] INRIA Lille – Nord Europe, ADAM Project-team,
University of Lille 1, LIFL CNRS UMR 8022,
F-59650 Villeneuve d'Ascq
{frederic.loiret,romain.rouvoy}@inria.fr

Abstract. The success of high-level programming models in *Wireless Sensor Networks* (WSNs) is heavily dependent on factors such as ease of programming, code well-structuring, degree of code reusability, and required software development effort. Component-based programming has been recognized as an effective approach to meet such requirements. Most of componentization efforts in WSNs were ineffective due to various reasons, such as high resource demand or limited scope of use. In this paper, we present REMORA, a new approach to practical and efficient component-based programming in WSNs. REMORA offers a well-structured programming paradigm that fits very well with resource limitations of embedded systems, including WSNs. Furthermore, the special attention to event handling in REMORA makes our proposal more practical for WSN applications, which are inherently event-driven. More importantly, the mutualism between REMORA and underlying system software promises a new direction towards separation of concerns in WSNs. Our evaluation results show that a well-configured REMORA application has an acceptable memory overhead and a negligible CPU cost.

Key words: Wireless sensor networks, component model, event-driven.

1 Introduction

The recent increase in the number and size of WSN applications makes *high-level programming* an essential need to the development of WSN platforms. However, this concept is still immature in the context of WSNs for various reasons. Firstly, the existing diversities in WSN hardware and software platforms have brought the same order of diversity to programming models for such platforms [1]. Moreover, developers' expertise in state-of-the-art programming models become useless in WSN programming as the well-established discipline of program specification is largely missing in this area. Secondly, the structure of programming models for WSNs are usually sacrificed for resource usage efficiency, thereby, the outcome of such models is usually a piece of tangled code maintainable only by its owner. Finally, application programming in WSNs typically requires learning low-level system programming languages, which imposes a significant burden on the programmer.

Software *componentization* has been recognized as a well-structured programming model able to tackle the above concerns. Separation of concerns, module reusability, controlling cohesion and coupling, and provision of standard API are some of the main features of *component-based software engineering* [2, 3]. Although using this paradigm in earlier embedded systems was relatively successful [4–7], most of the efforts in the context of WSNs remain inefficient or limited in the scope of use. TINYOS programming model, NESC [8],

is perhaps the most popular component model for WSNs. Whereas NesC eases WSN programming, this component model is tightly bound to the TinyOS platform. Other proposals, such as OpenCom [14] and THINK [20], are either too heavyweight for WSNs, or not able to support event-driven programming, which is of high importance in WSNs.

In this paper, we present Remora, a lightweight component model designed for resource-constraint embedded systems, including WSNs. The strong abstraction promoted by this model allows a wide range of embedded systems to exploit it at different software levels from *Operating System* (OS) to application. To achieve this goal, Remora provides a very efficient mechanism for event management, as embedded applications are inherently event-driven. Remora components are described in XML as an extension of the *Service Component Architecture* (SCA) model [10] in order to make WSN applications compliant with the state-of-the-art componentization standards. Additionally, the C-like language for component implementation in Remora attracts both embedded system programmers and PC-based developers to programming for WSNs. Finally, Remora features a coherent mechanism for component *instantiation* and *property-based component configuration* in order to facilitate lightweight event-driven programming in WSNs.

We demonstrate the promising result of deploying Remora components on Contiki—a leading operating system for WSNs [11]. The efficient use of Contiki features, such as process management and event distribution [12], on the one hand, and the abstraction layer linking Remora to Contiki, on the other hand, promise a very effective and generic approach towards practical high-level programming in WSNs.

The rest of the paper is organized as follows. In Section 2, the specification of the Remora component model is presented. Section 3 describes how Remora is implemented, while the evaluation results are reported in Section 4 including the assessment of a real Remora-based deployment. A survey of existing approaches and a discussion on Remora future work are presented in Section 5 and Section 6, respectively.

2 Remora Component Model

In this section, we first discuss the primary design concepts in Remora and then we explain the specifications of this component model. The design principles of Remora include:

XML-based Component Description. To achieve simplicity and generality, we adopt XML to describe components. The XML schema in Remora conforms to the *Service Component Architecture* (SCA) notations in order to accelerate standardization of component-based programming in WSNs. As SCA is originally designed for large-scale systems-of-systems, Remora extends SCA with its own architectural concerns to achieve realistic component-based programming in WSNs.

C-like Language for Component Implementation. Remora components are written in a C-like language enhancing the C language with features to support component-based and structured programming. This enhancement also attracts both embedded systems programmers and PC-based developers towards high-level programming in WSNs.

OS Abstraction Layer. The Remora component framework is integrated with underlying operating system through a well-defined OS-abstraction layer. This thin layer can easily be developed for all WSN operating systems supporting the C language like Contiki. This feature ensures portability of Remora components towards different OSs. The abstraction of Remora becomes more valuable when the component framework is easily configured to reuse OS-provided features, such as event processing and task scheduling.

Event Handling. Besides the support of events at operating system level in embedded systems, we also need to consider event handling at the application layer. Remora proposes a high-level support of event generation and event handling. Indeed, the event-processing model of Remora is one of its key features.

To describe our component model, we first define the basic terms used throughout this paper. Figure 1 illustrates the development process of Remora-based applications. A

REMORA application consists of a set of REMORA Components, containing descriptions and implementations of software modules. The REMORA *engine* processes the components and generates standard C code deployable within the REMORA *framework*. The framework is an OS-independent module supporting the specification of the REMORA component model. Finally, the REMORA application is deployed on the target sensor node through the REMORA *runtime*, which is an OS-abstraction layer integrating the application to the system software.

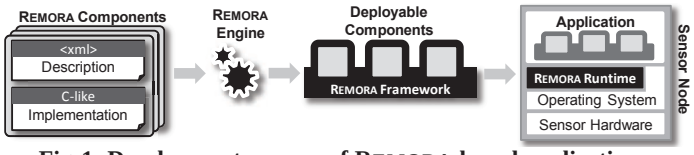

Fig. 1: Development process of REMORA-based applications.

2.1 Component Specification

A REMORA component contains two main artifacts: component *description* and component *implementation*. The component description is an XML document describing the specifications of the component including *services*, *references*, *producedEvents*, *consumedEvents*, and *properties*. A service describes the operations provided by the component, while a reference indicates the operations required by the component. Likewise, a producedEvent identifies an event type generated by a component, whereas a consumedEvent specifies component's interest on receiving a particular event. The component implementation is a C-like program containing three types of operations: *i)* operations implementing the component's services, *ii)* operations processing events, and *iii)* component's private operations.

To overview the REMORA specification, we first present the REMORA-based implementation of the traditional *blink* application, then we discuss REMORA features in details. Figure 2 depicts the components involved in this application which are in charge of *blinking* a LED on sensor node every three seconds.

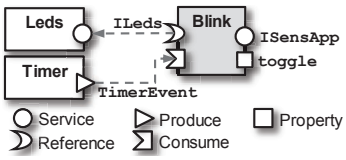

Fig. 2: A simple REMORA-based application.

We here focus on the Blink component and describe it according to the REMORA component model. Figure 3 shows the XML description of this component. Blink provides an `ISensorApp` interface to start application execution and requires an `ILeds` interface to switch LEDs on and off, which is implemented by the Leds component. It also exposes a property to `toggle` a LED on the sensor node. As Blink produces no event, the `producer` tag is empty, while it is subscribed to receive TimerEvent and process it in the `timerExpired` function.

```
<componentType name="app.BlinkApp">
  <service name="iSensorApp">
    <interface.remora name="core.boot.api.ISensorApp"/>
  </service>
  <reference name="iLeds">
    <interface.remora name="core.peripheral.api.ILeds"/>
  </reference>
  <property name="toggle" type="xsd:short">0</property>
  <producer/>
  <consumer operation="timerExpired">
    <event.remora type="core.sys.TimerEvent" name="aTimeEvent"/>
  </consumer>
</componentType>
```

Fig. 3: XML description of Blink component.

Figure 4 presents the excerpt of the Blink implementation. This C-like code implements the only function of the `ISensorApp` interface (`runApplication`) and handles TimerEvent within the `timerExpired` function. In the `runApplication` function, we specify that the TimerEvent generator (`aTimeEvent.producer`) is configured to generate periodically TimerEvent every three seconds. The last command in this function is used to notify the TimerEvent generator to start time measurement. When time is expired, Timer

sets the attributes of aTimeEvent (*e.g.*, latency) and then the REMORA framework calls the `timerExpired` function.

```
void runApplication(){
  aTimeEvent.producer.configure(3*CLOCK_SECOND, 1/*periodic*/);
  aTimeEvent.observation.start();
}
void timerExpired(){
  if (this.toggle == 0){
    iLeds.onLeds(LEDS_RED);
    this.toggle = 1;
  }else{
    iLeds.offLeds(LEDS_RED);
    this.toggle = 0;
  }
  printf("Time elapsed after interval: %d", aTimeEvent.latency);
}
```

Fig. 4: C-like implementation of Blink component.

Services and References. Components offer their function as *services* and may also depend on services provided by other components, so called *references*. A service consists of an *interface*, described in a separate XML with a name and the associated operations. Figure 5 presents the simplified `ILeds` interface used by the Blink component as a reference.

```
<interface.remora name="core.peripheral.api.ILeds">
  <operation name="getLeds" return="xsd:unsignedByte"/>
  <operation name="onLeds">
    <in name="leds" type="xsd:unsignedByte"/>
  </operation>
  <operation name="offLeds">
    <in name="leds" type="xsd:unsignedByte"/>
  </operation>
</interface.remora>
```

Fig. 5: A simplified description of `ILeds` interface.

Component Properties. Properties are the editable parameters provided by each component, converting components from a dead unit of functionality to an active entity tractable during the application lifespan. In particular, this enhancement occurs in event producer components, where we need to retain the state of the event producer to generate accurate events, *e.g.*, the Timer component in the Blink application. Properties also enable components to become either *stateless* or *stateful*. A component is stateful if and only if it defines a property, *e.g.*, the Blink component is stateful, while Leds is a stateless component.

Component Implementation. REMORA components are implemented by using a dialect of C language with a set of new commands. This C-like language is mainly proposed to support the unique characteristics of REMORA, namely, component instantiation, event processing, and property manipulation. Therefore, for pure component-based programming without the above features, the programmer can almost rely on C features. We implicitly introduced a few of these commands within the Blink component implementation, while the complete description of commands is available in [22].

2.2 Component Instantiation

Component instantiation is essentially proposed to manage efficiently event producer components. The REMORA engine greatly benefits from component instantiation when linking one producer to several consumers. For example, in the Blink application, the producer (Timer) of TimerEvent should be instantiated per consumer component, while the UserButtonEvent generator is a single-instance component publishing an event to all subscribed components when the user button on a sensor node is pressed.

Component instantiation is based on two principles: *i)* The component's code is always single-instance, and *ii)* the component's *context* is duplicated per new instance. By component context, we mean the *data structures* required to handle the properties independently from the component's code. Thus, a REMORA component becomes a *statically reconfigurable and reusable* entity and the memory overhead is kept very low by avoiding code duplication.

REMORA proposes three *multiplicity types* for the component' context: *raw-instance* (stateless component), *single-instance*, and *multiple-instances*. The REMORA engine features an algorithm determining the multiplicity type of a component based on: *i)* whether the component owns any property, *ii)* whether the component is an event producer, and *iii)* the number of components subscribed to a specific event. When the multiplicity type is determined, the REMORA engine statically allocates memory to each component instance.

2.3 Event Management

The REMORA design comprehensively supports event-based interactions between components. The event design principles in REMORA include:

Event Attributes. An event type in our approach can have a set of attributes with specific types. By defining attributes, the event producer can provide the event-specific information to the event consumer, *e.g.*, the `latency` attribute of TimerEvent in the Blink application.

Application Events vs OS Events. Events in REMORA are either *application-level events* or *OS-events*. Application events are generated by the REMORA framework (like Timer in the Blink application), while the latter are generated by OS. The REMORA runtime features mechanisms to observe OS-events, translate them into corresponding application-level events, and publish them through REMORA components.

Event Observation Interface. This interface is proposed to specify the time period during which events should be observed by producers, *e.g.*, the listening period of a TCP/IP event is the whole application lifespan (*automatic* observation), while a Timer event is observed according to the user-configured time (*manual* observation). REMORA proposes the *event observation interface* in order to control the manual observations. This generic interface includes operations, such as `start`, `pause`, `resume`, and `terminate`. If an event type is manually observable, the associated event producer should implement this interface. By doing that, the event consumer can handle the lifecycle of the observation process by calling operations in this interface without being aware of the associated event producer.

Event Configuration Interface. An event type can have an interface enabling the event consumer to configure event generation. Each component producing an event should implement the associated configuration interface identified in the specification of the event. This interface is designed to decouple completely the consumer and the producer.

Single Event Producer per Event Type. An event type in REMORA is produced by *one and only one* component. Instead of imposing the high overhead of defining event channels and binding manually event consumers and producers, the REMORA framework *autowires* producers and consumers. We believe that this constraint does not affect event-related requirements of applications. In case of having two producers generating one event type, we can define a new event type, extended from the original event, for one of the producers.

Event Casting. Events in our proposal can be either *unicast*, or *multicast*. Unicast is a one-to-one connection between an event producer and an event consumer (*e.g.*, TimerEvent), while a multicast event may be of interest to more than one component (*e.g.*, UserButtonEvent). The REMORA framework distinguishes between these two types in order to improve the efficiency of processing and distributing events. We also need to clear how multiplicity type of components on the one side, and unicast events and multicast events on the other side are related. To this end, we define two invariants:

Invariant1: *The consumer of a unicast event should be a raw-instance or single-instance component.*

Invariant2: *The producer of a multicast event should be a raw-instance or single-instance component.*

These invariants are mainly proposed to boost the efficiency of event processing in the REMORA framework. We do not support other event communication schemes since it implies to reify at runtime the source and the destination of an event and to maintain complex routing tables within the REMORA framework, which will induce significant overheads in

term of memory footprints and execution time. We believe these invariants do not limit event-related logic of embedded applications.

Events Description. Similar to components, events have their own descriptions, which are in accordance to the event specification in REMORA, discussed above. Figure 6 presents a simplified events description document of the Blink application. This document consists of two outer tags: `event.remora` and `event.os`, corresponding to the application events and the OS-events, respectively.

```
<eventType>
 <event.remora type="core.sys.TimerEvent" observation="manual" castType="unicast">
  <attribute name="latency" type="xsd:int"/>
  <configInterface>
   <operation name="configure">
    <in name="interval" type="xsd:int"/>
    <in name="periodic" type="xsd:short"/>
   </operation>
  </configInterface>
 </event.remora>
 <event.os/>
</eventType>
```

Fig. 6: Application events description.

2.3.1 Event Management Illustration

Figure 7 illustrates the event management mechanism implemented in REMORA. We explain the mechanism based on the steps labeled in the figure. During the first two steps, the event consumer can configure event generation and control event observation by calling the associated interfaces realized by the event producer component. These steps in our sample application are achieved in the Blink component (event consumer) by the code below:

```
aTimeEvent.producer.configure(3*CLOCK_SECOND, 1);
aTimeEvent.observation.start();
```

Note that the programmer is not aware of the TimerEvent producer. She/he only knows that the TimerEvent generator is expected to implement the `configure` function defined in the description of TimerEvent (cf. Figure 6). The TimerEvent producer should also implement the observation interface as the observation type of TimerEvent is manual.

Whereas the above steps are initiated by the programmer, the next two steps are performed by the REMORA framework. Step 3 is dedicated to *polling* the producer component to observe event occurrence. The event producer is polled by the REMORA framework through a *dispatcher* function in the producer. In fact, the event observation occurs in this function. The polling process is started, paused, resumed, and terminated based on the programmer's configuration for the event observation, performed in step 2.

For application-level events, the REMORA framework is in charge of calling periodically this function, while for OS-events, REMORA invokes this function whenever an OS-event is observed by the REMORA runtime. The REMORA runtime listens to only application-requested OS-events, and delivers the relevant ones to the framework. The REMORA framework then forwards the event to the corresponding OS-event producer component by calling its dispatcher function.

Finally, in step 4, upon detecting an event in the dispatcher function, the producer component creates the associated event, fills the required attributes, and publishes it to the REMORA framework. The framework in turn forwards the event to the interesting components by calling their event handler function.

2.4 Components Assembly and Deployment

A typical WSN application may contain several implementations of a certain component type due to the existing heterogeneity in such platforms. To configure an application according to the target platform, REMORA introduces components *assembly* (equivalent to *composite* component in SCA). This XML document lists the application components, as well as bindings between their references and services. Figure 8 shows the configuration of Blink application in which there is only one binding from Blink to the Leds component

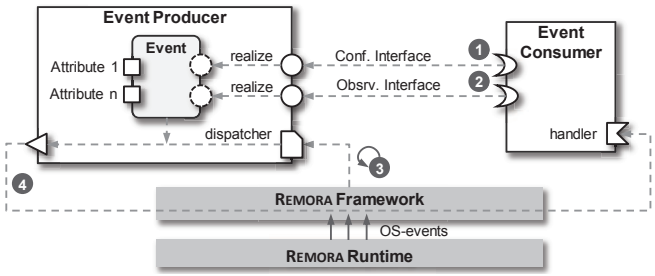

Fig. 7: Event management mechanism in REMORA.

implementing the `ILeds` interface for the MSP430 microcontroller. Note that the event-binding between Blink and Timer is created automatically by the REMORA framework.

```
<composite name="app.BlinkAppConfigurer">
  <component name="ledControl">
    <implementation.remora implementer="cmu.telosb.peripheral.Leds"/>
  </component>
  <component name="blink">
    <implementation.remora implementer="app.BlinkApp"/>
  </component>
  <component name="timer">
    <implementation.remora implementer="core.sys.Timer"/>
  </component>
  <wire source="blink/iLeds" target="ledControl/iLeds"/>
</composite>
```

Fig. 8: Blink application configuration.

Figure 9 demonstrates the four main phases of application deployment. The REMORA Development Box encompasses specification-supporting artifacts, as well as *External Types Definition*—a set of C header files containing application's type definitions. It should be noted that the component implementation can call OS libraries through a set of system APIs implemented by REMORA runtime components. Therefore, there is no hard-coded dependencies between REMORA implementers and the native API of the underlying OS. In the next phase, the REMORA engine reads the elements of the development box and also OS libraries in order to generate the REMORA framework including the source code of components and OS-support code (for deployment). Then, application object file will be created through OS-provided facilities and finally deployed on sensor nodes.

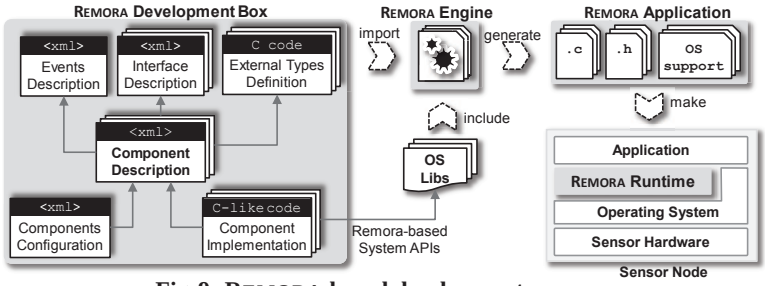

Fig. 9: REMORA-based development process.

3 Implementation

In this section, we discuss the key technologies, techniques, and methods used for the implementation of REMORA. We structure this section according to the phases proposed for REMORA-based application development.

3.1 REMORA Engine

The REMORA engine is designed to analyze the implementations of components and generate the equivalent C code, as well as OS-support code. The engine is written in Java because of its cross-platform capabilities, as well as its strong support for XML processing. Additionally, the object-oriented nature of Java simplifies the complex process of code analyzing and code generation. We briefly discuss the key design issues of the engine below.

The first concern of the REMORA engine is the mechanism for parsing the C-like implementation of components. To this end, we have developed a parser module, which is orig-

inally generated by ANTLR—a widely used open-source parser generator [13]. We have modified the generated parser to extract REMORA-required information, such as name, signature, and body of implementation functions.

Dealing with events, component instantiation and component configuration is the other key part of the REMORA engine. This unit deduces the multiplicity type of components and generates the necessary data structures. It also features a set of well-defined techniques, such as *in-component call graph analyzer* and *cross-component call tracker* to support stateful components. The former concept is concerned with discovering context-dependent functions of a component, and the latter tracks the interactions between components in order to retain the state of components. Finally, the major task of this part is to embed framework-support patches in the component implementation.

3.2 REMORA Framework

The REMORA framework is mainly designed to facilitate event management tasks, including *scheduling* and *dispatching*. To explain these tasks, we first introduce two *queue* data structures supporting our event model. The first queue is dedicated to the event producer components (PQ), while the second one is designed to maintain the event consumers (CQ). We discuss here how the REMORA framework is built based on these data structures.

Scheduling in REMORA refers to all arrangements required to *enqueue* and *dequeue* event producers and event consumers. In particular, the main concern is *when* to enqueue/dequeue a component and *who* should perform these tasks. The REMORA framework addresses these issues based on the observation model of events. For example, if an event is *automatically* observable, the associated producer component and all the subscribed consumers are enqueued by the framework core during the application startup, while in a *manual* observation, producer and consumer are placed respectively in PQ and CQ when the consumer component calls the `start` function of observation interface.

Figure 10 illustrates the *dispatching* mechanism in the framework including the supporting data structures. In *Polling*, the REMORA framework continuously polls the EventProducer components through *dispatcher*—the globally known callback function. Whenever a producer dispatches an event (AbstEvent), the framework casts this event to the actual event type, which is either UCastEvent(unicast event) or MCastEvent(multicast event). UCastEvent will be directly forwarded to the subscribed consumer through the callback function pointer stored in the UCastEvent. If a MCastEvent is generated, the framework delivers it to all the interesting components formerly enqueued. For OS-events, the same procedure is followed except the polling phase, which is performed by the operating system.

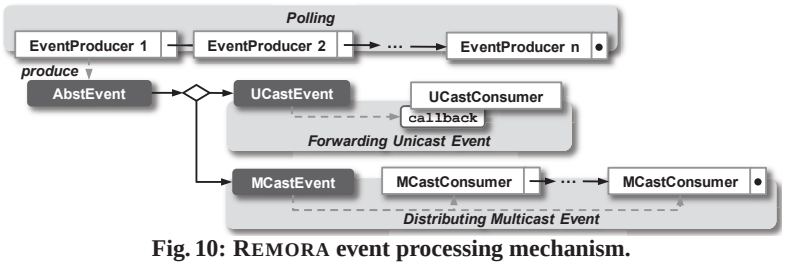

Fig. 10: REMORA event processing mechanism.

3.3 REMORA Runtime

The current implementation of the REMORA runtime is a Contiki-compliant *process* running together with all other *autostart* processes of Contiki. This process undertakes two tasks: *i)* periodically scheduling the REMORA framework (for polling event generator components) to run, and *ii)* listening to the OS-events and delivering the relevant ones to the REMORA framework. By relevant, we mean the REMORA runtime recognizes those OS-events that are of interest to the application. To achieve such filtering, the source code of this part is generated by the REMORA engine according to the events description (cf. Section 2.3) of

target application and then imported to the REMORA runtime. By doing that, we provide a lightweight event distribution mechanism interpreting only application-specific OS-events.

Additionally, the application code may need to use OS-provided libraries. REMORA proposes system API *wrapper* components for this purpose. In fact, these components delegate all high-level system calls to the corresponding OS-level functions, *e.g.*, the `currentTime()` function call in the system API is delegated to the Contiki function `clock_time()`. We offer this API to fully decouple the application components from OS modules and ensure the portability of REMORA. If an application is not expected to be ported to other platform types, the OS libraries can be directly called within the component implementation.

4 Evaluation

In this section, we first demonstrate and assess a real REMORA-based application, then we focus on the general performance figures of REMORA.

4.1 A Real REMORA-based Deployment

Our real application scenario is a network-level *application suite* consisting of a set of mini applications bundled together. This suite is basically designed to provide services, such as *code propagator* and *web facilities* in WSNs. We focus here on the first one and design it based on the REMORA approach.

Code propagation becomes a very important need in WSNs when we need to update remotely the running application's software [27]. The code propagator application is responsible for receiving all segments of a running application's object code over the network and loading the new application image afterwards. The code propagator exploits the TCP and UDP protocols to propagate code over the network. At first, TCP is used to transfer new code, block by block, to the sink node connected to the code repository machine, and then UDP is used to broadcast wirelessly new code from a sink node to other sensor nodes in the network. When all blocks are received, the code propagator loads the new application.

Figure 11 shows the components involved in the first part of our application scenario. TCPListener is a core component listening to TCP events. This multiple-instances event generator is created for each TCP event consumer component with unique listening port number. For example, CodePropagator receives data from port 6510 (`codePropPort`), while WebListener is notified for all TCPEvents on port 80 (`webPort`). CodePropagator stores all blocks of new code in the external flash memory through the `IFile` interface implemented by the FileSystem component. When all blocks are received, CodePropagator loads the new application by calling the `ILoader` interface from the ELFLoader component. These two interfaces are system APIs that delegate all application-level requests to the OS-specific libraries. The `INet` interface, implemented by the Network component, is also the other system API providing the low-level network primitives to TCPListener.

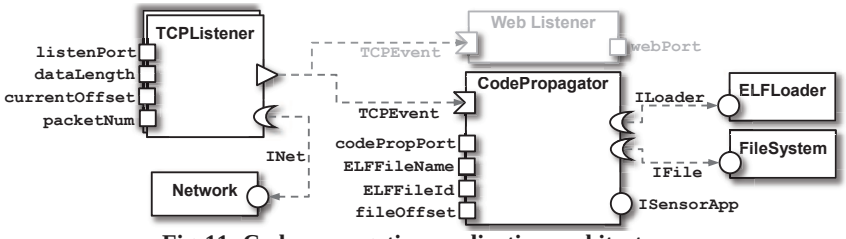

Fig. 11: Code propagation application architecture.

As mentioned before, we adopt Contiki as our OS platform to assess the REMORA component model. Contiki is being increasingly used in both academia and industrial applications in a wide range of sensor node types. Additionally, Contiki is written in the standard C language and hence REMORA can be easily ported to this platform. Finally, the great support of Contiki on event processing and process management motivate us to design and

implement the REMORA runtime on this OS. Our hardware platform is the popular TelosB mote equipped with a 16-bit TI MSP430 MCU with $48KB$ ROM and $10KB$ RAM.

The concrete separation of concerns in this application is the first visible advantage of using REMORA. The second improvement is the *easy* reuse of TCPListener for other TCP-required applications, which is not the case in a non-componentized implementation. In particular, for each new application, we only need to instantiate the *context* of TCPListener and configure its properties (like port number) accordingly, *e.g.*, WebListener in Figure 11.

Table 1 reports the memory requirement of REMORA and Contiki programming model (*protothreads*) for implementing the code propagation application. As indicated in the table, the REMORA-based development does not impose additional data memory overhead, while it consumes extra 532 bytes of code memory, which is essentially related to the cost of framework and runtime modules. This cost is paid once and for all, regardless of the size and the number of applications running on the sensor node. The code memory cost can be even further reduced by removing system APIs (Network, FileSystem, and ELFLoader) and calling directly the Contiki's libraries within CodePropagator. Note that the overhead of TCPListener can also be decreased when this component is shared for the use of other applications, *e.g.*, WebListener. Therefore, we can conclude that the memory overhead of REMORA is negligible compared to the high-level features it provides to the end-user.

Table 1: The memory requirement of code propagation application in REMORA-based and Contiki-based implementations.

Programming Model		**Code Memory (bytes)**	**Data Memory (bytes)**
Contiki		**722**	**72**
REMORA	Code Propagation Components		
	CodePropagator	252	36
	TCPListener	310	0
	System API Components		
	ELFLoader	38	0
	Network	92	0
	FileSystem	68	0
	REMORA Core		
	Framework and Runtime	494	14
	Total	**1254**	**50**
REMORA overhead		**+532**	**-22**

The rest of this section is devoted to the assessment of two main performance figures of REMORA, namely, memory footprints and CPU usage.

4.2 Memory Footprint

High memory usage has been one of the main reasons behind unsuccessfulness of component-based proposals for embedded systems. In REMORA, we have made a great effort to maintain memory costs as low as possible. The first step of this effort is to avoid creating meta-data structures, which are not beneficial in a static deployment. Distinguishing unicast events and multicast events has also led to a significant reduction in memory footprints as REMORA does not need to create any supporting data structure for unicast events.

The memory footprints in REMORA is categorized into a minimum overhead and a dynamic overhead. The former is paid once and for all, regardless of the amount of memory is needed for the application components, while the latter depends on the size of application. Table 2 shows the minimum memory requirements of REMORA, which turn out to be quite reasonable with respect to both code and data memory. As mentioned before, our sensor node, TelosB, is equipped with $48KB$ of program memory and $10KB$ of data memory. As Contiki consumes roughly $24KB$ (without µIP support) of both these memories, REMORA

has a very low memory overhead considering the provided facilities and the remaining space in the memory.

Table 3 shows the memory requirement of different types of modules in the REMORA framework. The exact memory overhead of REMORA depends on how an application is configured, *e.g.*, an application, containing one single instance event producer and one unicast event, needs extra 56 bytes ($38 + 8 + 10$) of both data and code memory. Ordinary components do not impose any memory overhead as REMORA does not create any meta data structures for them. For other types of modules, REMORA keeps the data memory overheads very low as this memory in our platform is really scarce. We also believe that the code memory overhead is not significant since a typical WSN application is small in size and it may contain up to a few tens of components, including ordinary components. It should be noted that componentization itself reduces the memory usage by maximizing the reusability degree of system functionalities like the one discussed in the code propagation application.

Table 2: The minimum memory requirement of REMORA.

Module	Code Memory (bytes)	Data Memory (bytes)
Framework Core	374	4
Runtime Core	120	10
Total	**494**	**14**

Table 3: The memory requirement of different entities in REMORA.

Entity		Code Memory (bytes)	Data Memory (bytes)
Ordinary Component		0	0
Event Producer	Single Ins.	38	8
	Multiple Ins.	42	10
Event	Unicast	0	10
	Multicast	0	10
Multicast Event Consumer		30	6
OS Event		28	4
System API		4	0

4.3 CPU Usage

As energy cost of REMORA core is limited to only the use of the processing unit, we focus on the processing cost of our approach and show that REMORA keeps the CPU usage at a reasonable level, and in some configurations it even reduces CPU usage compared to the Contiki-based application development.

To perform the evaluation, we set up a Blink application in which a varying number of mirror components (1 to 15) switch LEDs on and off every second. The two implementations of this application, Contiki-based and REMORA-based, were compared according to a CPU measurement metric. The metric was to measure the amount of time required by one REMORA component and one Contiki process to switch LEDs six times: three times on and three times off. With the less number of switches, we cannot extract the exact timing differences as our hardware platform provides a timing accuracy of the order of one millisecond.

We started our evaluation by deploying an application like the one presented in Section 2.1 and measuring the CPU usage based on our metric. In each next evaluation step, we added a mirror Blink component to the application and measured again the time. This experiment was continued for 15 times. We made the same measurement for a Contiki-based Blink application and added a new Contiki Blink process in each step. Figure 12 shows the evaluation result of our scenario. When we have one Blink component/process, the CPU overhead of both approaches is almost the same, indicating that the REMORA runtime and framework impose no additional processing overhead. When the number of components/process increases towards 15, reduction in CPU usage is achieved in two dimensions.

Firstly, the number of CPU cycles for REMORA is slightly less than for the Contiki application. This difference reaches 13 milliseconds when Contiki undertakes running 15 Blink processes. Therefore, we can conclude that REMORA does not impose additional processing overhead affecting the performance of the system. Secondly, the CPU usage of REMORA application is reduced when the number of Blink components is increased. This improve-

ment is achieved because the number of context switches between the REMORA runtime and the REMORA framework is significantly decreased when there are more event producer components (Timer) in PQ.

To clarify this issue, we assume that the application running time is T and Contiki periodically allocates CPU to the REMORA runtime in this period. In each allocation round, the runtime module invokes the event manager in the REMORA framework to poll the application level event producers. Given that there are K producers in PQ, the polling process consumes $K \times \mathrm{t}_1$ of CPU, where t_1 is the average processing cost of one element. Therefore, the frequency of event manager calling (equal to the number of context-switches) is in the order of $T/K \times \mathrm{t}_1$. Therefore, as the value of K is increased the number of context-switches is decreased accordingly. Figure 13 shows the changes in the number of context-switches when the number of Timer components is increased to 15. As a result, the maximum performance in REMORA relies on the average number of event producer components enqueued during the application lifespan, while in the worst case (a very few producers in the queue) REMORA does not impose any additional processing cost.

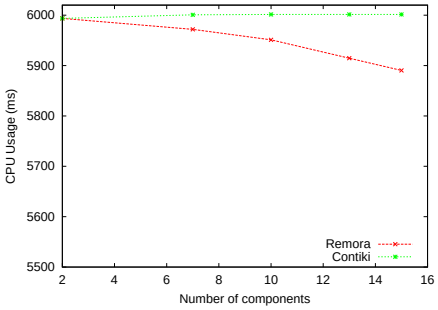

Fig. 12: The REMORA-based implementation does not impose additional CPU overhead compared to the Contiki-based implementation.

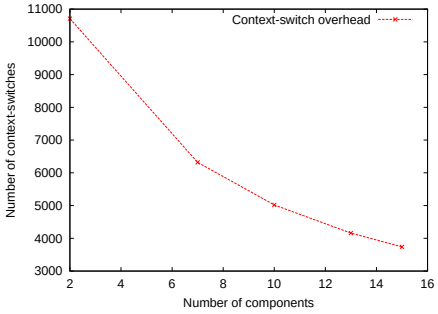

Fig. 13: As the number of producer components in the queue is increased, the number of context switches is significantly decreased.

5 Existing Approaches

In this section, we survey the existing component-based approaches for node-level programming on embedded system and WSNs. Most of these component models mainly aim at building entire operating systems as an assembly of components.

In the area of WSNs, NESC [8] is perhaps the best known component model being used to develop TINYOS [9]. As mentioned earlier, the main downside of NESC is that it is tightly bound to the TINYOS platform. Moreover, although NESC efficiently supports event-driven programming, events in NESC are not considered as independent entities with their own attributes and specifications. Therefore, the binding model of event-related components is not well-described as it is not essentially described based on the specification of events. Additionally, the unique features of REMORA, such as multiplicity in component instance and property-based reconfiguration of components bring significant improvements to component-based programming in WSNs compared to NESC.

Coulson et al. in [14] propose OPENCOM as a generic component-based programming model for building system applications without dependency on any target-specific platform environment. The authors express that they have tried to build OPENCOM with negligible overhead for supporting features specific to a development area, however it is a generic model and basically developed for platforms without resource constraints and tends to be complex for embedded systems. To evaluate OPENCOM, we deployed a sample *beacon* application [15], including Radio, Timer and Beacon components, on a TelosB node with Contiki. Based on our measurements, the memory footprint of this application is significantly high, so that it consumes $4,618$ bytes of code memory and 28 bytes of data memory.

The OSGi model [16] is a framework targeting powerful embedded devices, such as mobile phones and network gateways along with enterprise computers. OSGi features a secure execution environment, support for runtime reconfiguration, lifecycle management, and various system services. While OSGi is suitable for powerful embedded devices, the smallest implementation, Concierge [17] consumes more than $80KB$ of memory, making it inappropriate for resource-constrained platforms.

OSKIT [18] is a set of ready-made components for building operating systems. OSKIT is developed with a language called KNIT [19]. In contrast to NESC, KNIT is not limited to OSKIT. OSKit has adapted the Microsoft COM model and is not primarily focused on embedded systems.

The THINK framework [20] is an implementation of the FRACTAL [21] component model applied to operating systems. The choice of the THINK framework is motivated by the fact that it allows fine-grained reconfiguration of components. Although the experiments on deploying THINK components on WSNs have been quite promising in terms of memory usage [23], the lack of application-level event support is the main hurdle for using THINK in WSNs. LOOCI [24] is another component-based approach, providing a loosely-coupled component infrastructure focusing on an event-based binding model for WSNs. However, the Java-based implementation of LOOCI limits its usage to the SunSPOT sensor node.

6 Discussion, Conclusion and Future Direction

We presented REMORA, a novel programming abstraction for resource-constrained embedded systems. The main motivation behind proposing REMORA is to simplify high-level event-driven programming in WSNs by a component-based approach. Moreover, involving PC-based developers in WSN programming and considering the state-of-the-art technologies for component development are two other challenges addressed by REMORA. The special consideration paid to the event abstraction in REMORA makes it a practical and efficient approach for WSN applications development. The other key features of REMORA include: applicability on a wide range of embedded OSs, rich support of component reusability and instantiation, and reduced effort and resource usage in WSN programming.

Careful restrictions on the REMORA component model, including the lack of dynamic memory allocation and avoiding M-to-N communications between event producers and event consumers bring significant improvements to the static deployments in WSNs. Since one of our main future directions is to support dynamic component reconfiguration in REMORA [25–27], we encounter a new major challenge on how to efficiently provide such a feature in REMORA so that the overhead of dynamic memory allocation is carefully minimized.

As mentioned earlier, the current goal of REMORA is to be exploited only in application-level programming. However, we believe that the efficient support of event processing in REMORA potentially enables it to componentize system level functionalities. In the Blink application, we implicitly demonstrated this capability by redeveloping the Timer component, which is essentially developed at the OS level. To address precisely this issue, we need to enhance the current REMORA implementation with features like *concurrency support*, *task scheduling*, and *interrupts handling*.

In our current implementation, a REMORA process cannot be preempted by any other process in the operating system. This issue becomes critical when a component execution takes a long time to complete and it causes large average waiting times for other processes waiting for the CPU. The event handling model of REMORA can be used to provide preemption by defining a new event type per preemption-required point of application, while in this case the component implementation and the event management become quite complicated. This concern will also be considered in the future extensions for REMORA. In particular, we intend to promote the native Contiki macros, handling process lifecycle, to the REMORA application level. In this way, the REMORA component becomes preemptable by explicitly yielding the running process.

Beside the fact that REMORA provides a strong abstraction for single node programming, the same level of programming abstraction is expected to occur at the network level. This challenge opens up another key area for future work: how to make REMORA components distributed by the provision of a well-defined remote invocation mechanism.

Acknowledgments. This work was partly funded by the Research Council of Norway through the project SWISNET, grant number 176151.

References

1. Sugihara, R., Gupta, R.K.: Programming models for sensor networks: A survey. ACM. Trans. Sensor Networks 4(2), 1-29 (2008)
2. C. Szyperski. Component Software: Beyond Object-Oriented Programming. Second edition, ACM, Press and Addison-Wesley, New York, N.Y. (2002)
3. F. Bachmann, L. et al.: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Carnegie Mellon Software Engineering Institute (2000)
4. Ommering, R., Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software, IEEE Computer, vol. 33, no. 3 (2000)
5. Winter, M. et al.: Components for embedded software: the PECOS approach. In Proc. of the CASES'02, ACM Press, NY (2002)
6. Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M.: SaveCCM-a component model for safety-critical real-time systems. In Proc. of the IEEE Euromicro Conference (2004)
7. Plsek, A., Loiret, F., Merle, P., Seinturier, L.: A Component Framework for Java-Based Real-Time Embedded Systems. In Proc. of the ACM/IFIP/USENIX 9th Middleware (2008)
8. Gay, D. et al.: The nesC Language: A Holistic Approach to Networked Embedded Systems, In Proc. of the SIGPLAN Conference on Prog. Language Design and Impl. (2003)
9. Levis, P. et al.: TinyOS: An Operating System for Sensor Networks. Ambient Intelligence (2005)
10. `http://www.oasis-opencsa.org/sca`
11. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors, in Proc. of 1st Wkshp. on Embedded Networked Sensors (2004)
12. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems, Proc. ACM SenSys, (2006)
13. ANTLR. Website: `http://www.antlr.org`
14. Coulson, G. et al.: A generic component model for building systems software. ACM Trans. Computer Systems, 1-42 (2008)
15. WISEBED. `http://www.wisebed.eu/wiki/pmwiki.php?n=Main.Osaapp1`
16. The OSGi Alliance. The OSGi framework. http://www.osgi.org, 1999.
17. Rellermeyer, J., Alonso, G., Concierge: A Service Platform for Resource-Constrained Devices, in ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, June 2007, pp. 245 - 258
18. Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O.: The Flux OSKit: A Substrate for Kernel and Language Research, Operating Systems Principles (1997)
19. Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E.: Knit: Component Composition for Systems Software, Operating Systems Design and Implementation (OSDI) (2000)
20. Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G.: Think: A software framework for component-based operating system kernels. In Proc. of the USENIX Annual Conference (2002)
21. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J., B.: The FRACTAL component model and its support in Java. Softw., Pract. Exper. (2006)
22. REMORA. Website: `http://folk.uio.no/amirhost/remora`
23. Lobry, O., Navas, J., Babau, J.: Optimizing Component-Based Embedded Software, 2nd IEEE Workshop on Component-Based Design of Resource-Constrained Sys., COMPSAC-09, (2009)
24. Hughes, D. et al.: LooCI: A loosely-coupled component infrastructure for networked embedded systems, Mobile computing Multimedia, (2009)
25. Taherkordi, A. et al.: WISEKIT: A Distributed Middleware to Support Application-level Adaptation in Sensor Networks, In Proc. of DAIS'09, LNCS vol. 5523, Portugal, (2009)
26. Taherkordi, A., Rouvoy, R., Le-Trung, Q., Eliassen, F.: A Self-Adaptive Context Processing Framework for Wireless Sensor Networks, In Proc. of ACM MidSens'08, Belgium, (2008)
27. Mottola, L. et al.: Selective Reprogramming of Mobile Sensor Networks through Social Community Detection, In Proc. of EWSN'10, Portugal, (2010)